Embedded Real-Time Video Encryption Module on UAV Surveillance Systems

CIPRIAN RĂCUCIU, NICOLAE JULA, CONSTANTIN BĂLAN, COSMIN ADOMNICĂI
Communications Department
Military Technical Academy
B-dul George Coşbuc 81-83, Sector 5, Bucharest, ROMANIA
ciprian.racuciu@gmail.com, nicolae.jula@gmail.com, constantin.balan@yahoo.com, ado_atm@yahoo.com

Abstract: - A real-time video encryption module, developed to be used on an electrical accelerated mini-helicopter (UAV), is presented in this paper. The surveillance system is composed from six modules: image capture, encryption, two radio link modules, decryption and display module. The hart of the encryption system is the CV700C motherboard and it's VIA C7 microprocessor with ultra low power consumption and efficient heat dissipation. For encryption, Rijndael algorithm was used.

Key-Words: - Encryption, Rijndael, DirectShow, Real-time, Wireless, Embedded

1 Introduction

An UAV[1] is an airborne system, without a pilot, which flies by means of a remote control or by using an autopilot installed on board and carries sensors or weapons. It can be used only once or reused many times. Comparing an UAV with a classical airplane, most of the times, the UAV is small and light and the load is composed from sensors used in reconnaissance and surveillance missions, or in target acquisition missions. Now, there are new mission for UAV, like combat, intelligence, and civil applications [6]. Because any transmission from a military UAV must be secured it's a must to use an encryption method to protect de confidentiality of the transmitted data. Because of the restrictions applied to weight, dimensions and power consumption of any UAV module, it was chosen the CV700C motherboard manufactured by Lex Inc. which is 200 mm long and 150 mm wide. For an increased endurance to vibrations, the hard disk was replaced with a Compact Flash memory card. To maximize the encryption speed and to reduce the footprint the Windows XP Embedded operating system was used. The program was written in C++ programming language using the DirectShow API[2] which has the whole support for buffering and frame dropping.

2 Problem Formulation

Because any transmission from a military UAV must be secured it's a must to use an encryption method to protect de confidentiality of the transmitted data. Because of the restrictions applied to weight, dimensions and power consumption of any UAV module, it was chosen the CV700C motherboard manufactured by Lex Inc. which is 200 mm long and 150 mm wide. For an increased endurance to vibrations, the hard disk was replaced with a Compact Flash memory card. To maximize the encryption speed and to reduce the footprint the Windows XP Embedded operating system was used [13]. The program was written in C++ programming language using the DirectShow API[3] which has the whole support for buffering and frame dropping.

3 Problem Solution

The system is composed of:

- Digital video camera with USB interface;
- CV700C motherboard;
- RF modules;
- Notebook.

Device	Weight[g]	Price[€]
CV700C	500g	320€
Gigabyte 802.11b/g	15g	20€
Canion video	20g	10€

Table 1 Weight and price of used components

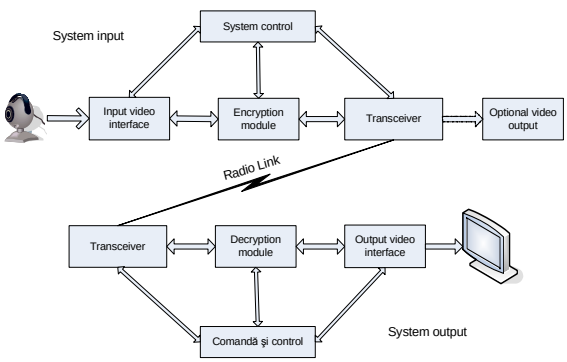

Fig. 1 The system

[1] Unmanned Aerial Vehicle
[2] Application Interface
[3] Application Interface

Fig. 2 CV 700C motherboard

The motherboard is based on VIA C7 microprocessor which operates at a frequency of 1 GHz. The microprocessor is built for applications with low power consumption requirements. On full load, the consumption of the microprocessor is 11 W and of the entire board is 25 W. To the motherboard are connected a video camera which takes pictures at a rate of 25 frames/second and an 802.11 wireless radio module which operates in the public spectrum. On this motherboard runs the airborne encryption subsystem. The other component of the system is the base station which runs on an Intel based microprocessor notebook. It receives data from the integrated 802.11 wireless module and feeds the data to the decryption module and displays the decrypted images on the monitor. The airborne encryption module can be controlled via radio link and there can be changed the encryption keys and the encryption modes.

Fig. 3 An UAV built by Professor Nicolae Jula

Fig. 4 Attached video camera details

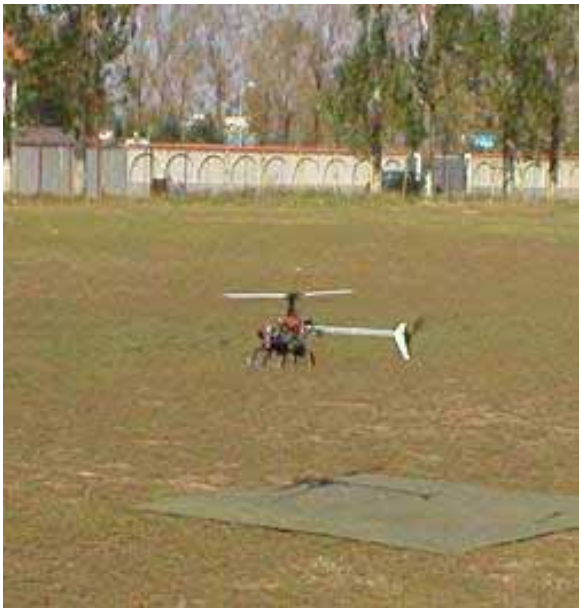

Fig. 5 Aspect of an UAV flying

3.1 The operating system

Windows XP Embedded is a modular version Windows XP Professional and the developer can choose which components are included in the distribution, making the Windows XP operating system flexible to developer's specific needs.

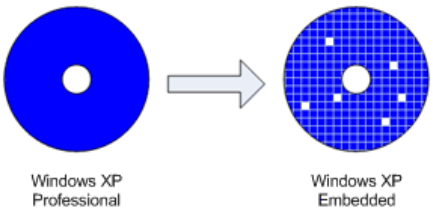

Fig. 6 Windows XP Professional versus Windows XP Embedded

The advantages of choosing which components to install are speed and a small footprint. In the case of a small footprint the mechanical hard drive can be

replaced with a Compact Flash memory card, making the system more resilient to vibrations. This will also reduce the power consumption of the system which is very important in the case of a battery driven system. Also, eliminating useless components, drivers, programs and services that slow down the operating system can leave free resources available to the encryption software.
A particular set of components that Windows XP Embedded has and Windows XP Professional doesn't is the Enhanced Write Filter which is designed especially for Compact Flash drives.

Enhanced Write Filter is composed of three components:

- EWF Manager Console application;
- Enhanced Write Filter;
- EWF NTLDR.

EWF reduces the number of write operations to the minimum. This is done by implementing a RAM buffer of the write operations. So, only one operations of writing operation is done for many write operation requested.

The first component, EWF Manager Console application, is the configuration console for EWF. The second component, EWF, is the core of the module. The third component, EWF NTLDR, is the boot loader used for a EWF configured drive.
The only disadvantage of EWF is the extra RAM needed for it.

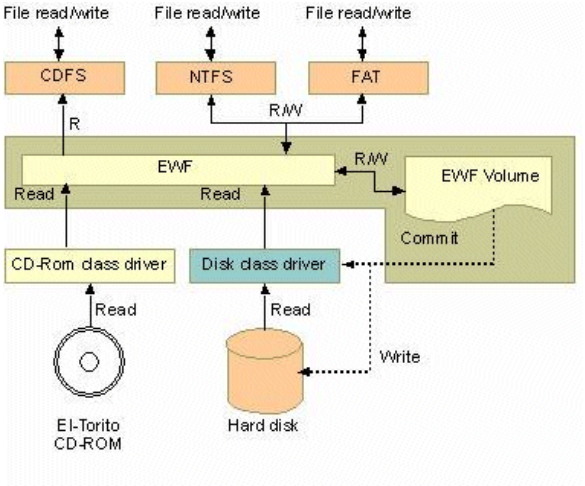

Fig. 7 EWF overview

3.2 DirectShow API

To benefit from the full power of Windows operating system, the DirectShow framework was used.
DirectShow is part of Microsoft Platform SDK and it is the most complex API that Microsoft ever created.
The DirectShow framework is a filter based framework, in which the processing modules are organized in a filter graph. Every processing module is named a filter. There can be created several independent filter graphs, which can or cannot be linked. Every multimedia device that is installed in a Windows environment can be accessed through the DirectShow API. Basically, every filter is a COM object which has to obey specific rules. Depending on the filter type, there are methods that have to be implemented. The developed has only the task of writing the code used for video processing only. The rest is leaved in the hands of the API. The synchronization between the filters, frame buffering, frame dropping are done by the API [10].

The basic power and flexibility of DirectShow derives directly from its modular design. DirectShow defines a standard set of Component Object Model (COM) interfaces for filters and leaves it up to the programmer to arrange these components in some meaningful way. Filters hide their internal operations; the programmer doesn't need to understand or appreciate the internal complexities of the Audio Video Interleaved (AVI) file format, for example, to create an AVI file from a video stream. All that's required is the appropriate sequence of filters in a filter graph.

Filters are atomic objects within DirectShow, meaning they reveal only as much of themselves as required to perform their functions.
Because they are atomic objects, filters can be thought of and treated just like puzzle pieces. The qualities that each filter possesses determine the shape of its puzzle piece, and that, in turn, determines which other filters it can be connected to. As long as the pieces match up, they can be fitted together into a larger scheme, the filter graph.

All DirectShow filters have some basic properties that define the essence of their modularity. Each filter can establish connections with other filters and can negotiate the types of connections it's willing to accept from other filters. A filter designed to process MP3 audio doesn't have to accept a connection from a filter that produces AVI video and probably shouldn't. Each filter can receive some basic messages run, stop and pause that control the execution of the filter graph. That's about it; there's not much more a filter needs to be ready to go. As long as the filter defines these properties publicly through COM, DirectShow will treat it as a valid element in a filter graph.

This modularity makes designing custom DirectShow filters a straightforward process. The programmer's job is to design a COM object with the common interfaces for a DirectShow filter, plus whatever custom processing the filter requires.

The modularity of DirectShow extends to the filter graph. Just as the internals of a filter can be hidden from the programmer, the internals of a filter graph can be hidden from view. When the filter graph is treated as a module, it can assume responsibility for connecting filters together in a meaningful way. It's possible to create a complete, complex filter graph by adding a

source filter and a renderer filter to the filter graph. These filters are then connected with a technique known as Intelligent Connect. Intelligent Connect examines the filters in the filter graph, determines the right way to connect them, adds any necessary conversion filters, and makes the connections all without any intervention from the programmer. Intelligent Connect can save an enormous amount of programming time because DirectShow does the tedious work of filter connection.

There is a price to be paid for this level of automation: the programmer won‘t know exactly which filters have been placed into the filter graph or how they‘re connected. With Intelligent Connect, the programmer won‘t know which filter DirectShow has chosen to use (at least, when a choice is available). It‘s possible to write code that will make inquiries to the filter graph and map out the connections between all the filters in the filter graph, but it ‘s more work to do that than to build the filter graph from scratch. So, modularity has its upsides ease of use and extensibility and its downsides hidden code.

Filters are the basic units of DirectShow programs, the essential components of the filter graph. A filter is an entity complete unto itself. Although a filter can have many different functions, it must have some method to receive or transmit a stream of data. Each filter has at least one pin, which provides a connection point from that filter to other filters in the filter graph. Pins come in two varieties: input pins can receive a stream, while output pins produce a stream that can be sent along to another filter.

There are three basic classes of DirectShow filters, which span the path from input, through processing, to output (or, as it‘s often referred to, rendering). All DirectShow filters fall into one of these broad categories. A filter produces a stream of data, operates on that stream, or renders it to some output device.

Any DirectShow filter that produces a stream is known as a source filter. The stream might originate in a file on the hard disk, or it might come from a live device, such as a microphone, webcam, or digital camcorder. If the stream comes from disk, it could be a pre-recorded WAV (sound), AVI (movie), or Windows Media file. Alternately, if the source is a live device, it could be any of the many thousands of Windows-compatible peripherals. DirectShow is closely tied in to the Windows Driver Model (WDM), and all WDM drivers for installed multimedia devices are automatically available to DirectShow as source filters. So, for example, webcams with properly installed Windows drivers become immediately available for use as DirectShow source filters. Source filters that translate live devices into DirectShow streams are known as capture source filters.

Transform filters are where the interesting work gets done in DirectShow. A transform filter receives an input stream from some other filter (possibly a source filter), performs some operation on the stream, and then passes the stream along to another filter. Nearly any imaginable operation on an audio or video stream is possible within a transform filter. A transform filter can parse (interpret) a stream of data, encode it (perhaps converting WAV data to MP3 format) or decode it, or add a text overlay to a video sequence. DirectShow includes a broad set of transform filters, such as filters for encoding and decoding various types of video and audio formats.
Transform filters can also create a tee in the stream, which means that the input stream is duplicated and placed on two (or more) output pins. Other transform filters take multiple streams as input and multiplex them into a single stream. Using a transform filter multiplexer, separate audio and video streams can be combined into a video stream with a soundtrack.

A renderer filter translates a DirectShow stream into some form of output. One basic renderer filter can write a stream to a file on the disk. Other renderer filters can send audio streams to the speakers or video streams to a window on the desktop. The Direct in DirectShow reflects the fact that DirectShow renderer filters use DirectDraw and DirectSound, supporting technologies that allow DirectShow to efficiently pass its renderer filter streams along to graphics and sound cards. This ability means that DirectShow‘s renderer filters are very fast and don ‘t get tied up in a lot of user-to-kernel mode transitions. (In operating system parlance, this process means moving the data from an unprivileged level in an operating system to a privileged one where it has access to the various output devices.)

A filter graph can have multiple renderer filters. It is possible to put a video stream through a tee, sending half of it to a renderer filter that writes it to a file, and sending the other half to another renderer filter that puts it up on the display. Therefore, it is possible to monitor video operations while they‘re happening, even if they‘re being recorded to disk.

All DirectShow filter graphs consist of combinations of these three types of filters, and every DirectShow filter graph will have at least one source filter, one renderer filter, and (possibly) several transform filters. In each filter graph, a source filter creates a stream that is then operated on by any number of transform filters and is finally output through a renderer filter. These filters are connected together through their pins, which provide a well-defined interface point for transfer of stream data between filters. Although every DirectShow filter has pins, it isn’t always possible to connect an input pin to an output pin. When two filters are connecting to each other, they have to reach an agreement about what kind of stream data

they‘ll pass between them.

The pins on a DirectShow filter handle the negotiation between filters and ensure that the pin types are compatible before a connection is made between any two filters. Every filter is required to publish the list of media types it can send or receive and a set of transport mechanisms describing how each filter wants the stream to travel from output pin to input pin.

When a DirectShow filter graph attempts to connect the output pin of one filter to the input pin of another, the negotiation process begins. The filter graph examines the media types that the output pin can transmit and compares these with the media types that the input pin can receive. If there aren’t any matches, the pins can‘t be connected and the connection operation fails.

Next the pins have to agree on a transport mechanism. Once again, if they can‘t agree, the connection operation fails. Finally one of the pins has to create an allocator, an object that creates and manages the buffers of stream data that the output pin uses to pass data along to the input pin. The allocator can be owned by either the output pin or the input pin; it doesn‘t matter, so long as they‘re in agreement.

If all these conditions have been satisfied, the pins are connected. This connection operation must be repeated for each filter in the graph until there‘s a complete, uninterrupted stream from source filter, through any transform filters, to a renderer filter. When the filter graph is started, a data stream will flow from the output pin of one filter to the input pin of the other through the entire span of the filter graph.

One of the greatest strengths of DirectShow is its ability to handle the hard work of supporting multiple media formats. Most of the time it‘s not necessary for the programmer to be concerned with what kinds of streams run through a filter graph. Yet to connect two pins, DirectShow filters must have clear agreement on the media types they‘re handling. Intelligent Connect automates the connection process between two pins. Two pins can be connected directly, so long as their media types agree. In a situation in which the media types are not compatible, the programmer often need one (or several) transform filters between the two pins so that they can be connected together. Intelligent Connect does the work of adding and connecting the intermediate transform filters to the filter graph.

For example, a filter graph might have a source filter that produces a stream of DV data perhaps it‘s connected to a camcorder. This filter graph has a renderer filter that writes a file to disk. These two filters have nothing in common. They don‘t share any common media types because the DV data is encoded and interleaved and must be decoded and de-interleaved before it can be written to a file. With Intelligent Connect, the filter graph can try combinations of intermediate transform filters to determine whether there‘s a way to translate the output requirements of the pin on the source filter into the input requirements of the render filter. The filter graph can do this because it has access to all possible DirectShow filters. It can make inquiries to each filter to determine whether a transform filter can transform one media type to another which might be an intermediate type transform that type into still another, and so on, until the input requirements of the renderer filter have been met. A DirectShow filter graph can look very grotesque by the time the filter graph succeeds in connecting two pins, but from the programmer‘s point of view, it‘s a far easier operation. And if an Intelligent Connect operation fails, it‘s fairly certain there‘s no possible way to connect two filters. The Intelligent Connect capability of DirectShow is one of the ways that DirectShow hides the hard work of media processing from the programmer.

The DirectShow filter graph organizes a group of filters into a functional unit. When connected, the filters present a path for a stream from source filters, through any transform filters, to renderer filters. However, it isn’t enough to connect the filters; the filter graph has to tell the filters when to start their operation, when to stop, and when to pause. In addition, the filters need to be synchronized because they‘re all dealing with media samples that must be kept in sync. For this reason, the filter graph generates a software-based clock that is available to all the filters in the filter graph. This clock is used to maintain synchronization and allows filters to keep their stream data in order as it passes from filter to filter. Available to the programmer, the filter graph clock has increments of 100 nanoseconds. (The accuracy of the clock on the system might be less precise than 100 nanoseconds because accuracy is often determined by the sound card or chip set on the system.)

When the programmer issues one of the three basic DirectShow commands run, stop, or pause the filter graph sends the messages to each filter in the filter graph. Every DirectShow filter must be able to process these messages.

For example, sending a run message to a source filter controlling a webcam will initiate a stream of data coming into the filter graph from that filter, while sending a stop command will halt that stream. The pause command behaves superficially like the stop command, but the stream data isn‘t cleared out like it would be if the filter graph had received a stop command. Instead, the stream is frozen in place until the filter graph receives either a run or stop command. If the run command is issued, filter graph execution continues with the stream data already present in the filter graph when the pause command was issued.

There were developed several filters:

- Encryption and decryption filter;
- Data link interface modules (client and server);

For video capture and video rendering pre-programmed filters were used.

The video input and output filters are contained in the DirectShow API. The main program initialises the filters and controls different parameters like:

- Image resolution
- Bit depth
- Encryption mode;
- Encryption key;

In this case, a resolution of 320x240 pixels and a bit depth of 24 bits/pixel were used. So the entire image contained 230400 bytes without the header used to pass the image parameters between filters, which added extra data [8].

3.3 The encryption module

Rijndael is a symmetric block cipher that can process data blocks of 128, 192 or 256 bits, using cipher keys with lengths of 128, 192 or 256 bits. In the AES standard, the block cipher can only process data blocks of 128 bits and the key lengths are 128, 192 or 256 bits [7].

An intermediate result of the encryption process is called "State". The "State" can be pictured as a rectangular array of bytes with four rows and Nb columns. If N is the block size, then Nb = N/32.

The Cipher Key is similarly pictured as a rectangular array with four rows. If M is the key size, then the number of columns is Nk = M/32.

$a_{0,0}$	$a_{0,1}$	$a_{0,2}$	$a_{0,3}$
$a_{1,0}$	$a_{1,1}$	$a_{1,2}$	$a_{1,3}$
$a_{2,0}$	$a_{2,1}$	$a_{2,2}$	$a_{2,3}$
$a_{3,0}$	$a_{3,1}$	$a_{3,2}$	$a_{3,3}$

Fig. 8 Example of State (with Nb=4)

$k_{0,0}$	$k_{0,1}$	$k_{0,2}$	$k_{0,3}$
$k_{1,0}$	$k_{1,1}$	$k_{1,2}$	$k_{1,3}$
$k_{2,0}$	$k_{2,1}$	$k_{2,2}$	$k_{2,3}$
$k_{3,0}$	$k_{3,1}$	$k_{3,2}$	$k_{3,3}$

Fig. 9Example of the Cipher Key (with Nk=4)

Each individual element of the matrices shown in Fig.8 and Fig. 9 is 4-byte vector or word.

In the actual algorithm, the input and output at the external interface are considered to be one-dimensional array of 8 bytes numbered upwards from 0 to 4*Nb-1. These blocks have lengths of 16 bytes and array index in the range 0…15.

The cipher key is considered to be a one-dimensional array of 8 bytes numbered upwards from 0 to 4*Nk-1. These blocks have lengths of 16, 24 or 32 bytes and array indices in the ranges 0…15, 0…23 or 0…31.

If the one-dimensional array index of a byte within block and the two dimensional index is (i, j), we have:

$$i=n \bmod 4$$
$$j=[n/4]$$
$$n=i+4*j$$

The number of rounds is denoted by Nr. The value of Nr depends of Nb and Nk and it is shown in Table 2.

Nr	Nb=4
Nk=4	10
Nk=6	12
Nk=8	14

Table 2 Number of rounds Nr

At the start of the Cipher, the input is copied into the State array. After an initial Round Key addition, the State array is transformed by implementing a round function for Nr times, with the final round differing slightly from the first Nr-1 rounds. The final State is then copied to the output.

The encryption module is based on the Rijndael algorithm in which the block and key size are limited at 128 bits. This limitation appeared to meet the encryption speed requirements. The algorithm is composed from four major operations made to the data block:

- SubBytes
- ShiftRows
- MixCollumns
- AddRoundKey

Rijndael algorithm uses the finite field $GF(2^8)$. The input bytes are considered polynomials and the addition, multiplication and other operations are done in $GF(2^8)$. Without improvements, these operations would take a lot of time to compute and the algorithm could not be used in a real-time environment. In FIPS-197, logarithms tables are provided to simplify the multiplication operations. Instead of working from the low order terms, and repeatedly multiplying by x^1 the speed can be improved by table lookup and polynomial addition:

```
public byte FFMul(unsigned byte a, unsigned byte b)
{
int t = 0;;
if (a == 0 || b == 0) return 0;
t = L[a] + L[b];
if (t > 255) t = t - 255;
return E[t];
}
```

The SubBytes transformation is defined as a non-linear

byte substitution which operates on the State. It's composed of two transformations:

- The multiplicative inverse in GF(2^8) is taken;
- The result is modified according to the fallowing transformation:

$$\begin{bmatrix} y_0 \\ y_1 \\ y_2 \\ y_3 \\ y_4 \\ y_5 \\ y_6 \\ y_7 \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 & 0 & 1 & 1 & 1 & 1 \\ 1 & 1 & 0 & 0 & 0 & 1 & 1 & 1 \\ 1 & 1 & 1 & 0 & 0 & 0 & 1 & 1 \\ 1 & 1 & 1 & 1 & 0 & 0 & 0 & 1 \\ 1 & 1 & 1 & 1 & 1 & 0 & 0 & 0 \\ 0 & 1 & 1 & 1 & 1 & 1 & 0 & 0 \\ 0 & 0 & 1 & 1 & 1 & 1 & 1 & 0 \\ 0 & 0 & 0 & 1 & 1 & 1 & 1 & 1 \end{bmatrix} \begin{bmatrix} x_0 \\ x_1 \\ x_2 \\ x_3 \\ x_4 \\ x_5 \\ x_6 \\ x_7 \end{bmatrix} + \begin{bmatrix} 1 \\ 1 \\ 0 \\ 0 \\ 0 \\ 1 \\ 1 \\ 0 \end{bmatrix}$$

Fig. 10 SubBytes transformation

For AES the irreducible polynomial is $x^8+x^4+x^3+x+1$. This polynomial is used to generate GF(2^8) used in finding the multiplicative inverse mentioned above.
The non-linearity of this transformation comes from multiplicative inverse in GF(2^8). Everything else in this transformation is linear.

In SubBytes, the calculation of the multiplicative inverse can be efficiently done using a "table lookup" method: a small table of 28 = 256 pairs of bytes can be built once and used forever (the table can be "hardwired" into hardware or software implementations). In this table of pairs, the zero byte is paired with the zero byte; the rest of the 255 entries in the table are the 255 cases of the pair (x, x^{-1}) where inversion is performed in the field. The "table lookup" method not only is efficient, but also prevents a timing analysis attack which is based on observing the operation time difference for different data which may suggest whether an operation is performed on bit 0 or bit 1. The "table lookup" method can actually include the whole transformation altogether.

x \ y	0	1	2	3	4	5	6	7	8	9	a	b	c	d	e	f
0	63	7c	77	7b	f2	6b	6f	c5	30	01	67	2b	fe	d7	ab	76
1	ca	82	c9	7d	fa	59	47	f0	ad	d4	a2	af	9c	a4	72	c0
2	b7	fd	93	26	36	3f	f7	cc	34	a5	e5	f1	71	d8	31	15
3	04	c7	23	c3	18	96	05	9a	07	12	80	e2	eb	27	b2	75
4	09	83	2c	1a	1b	6e	5a	a0	52	3b	d6	b3	29	e3	2f	84
5	53	d1	00	ed	20	fc	b1	5b	6a	cb	be	39	4a	4c	58	cf
6	d0	ef	aa	fb	43	4d	33	85	45	f9	02	7f	50	3c	9f	a8
7	51	a3	40	8f	92	9d	38	f5	bc	b6	da	21	10	ff	f3	d2
8	cd	0c	13	ec	5f	97	44	17	c4	a7	7e	3d	64	5d	19	73
9	60	81	4f	dc	22	2a	90	88	46	ee	b8	14	de	5e	0b	db
a	e0	32	3a	0a	49	06	24	5c	c2	d3	ac	62	91	95	e4	79
b	e7	c8	37	6d	8d	d5	4e	a9	6c	56	f4	ea	65	7a	ae	08
c	ba	78	25	2e	1c	a6	b4	c6	e8	dd	74	1f	4b	bd	8b	8a
d	70	3e	b5	66	48	03	f6	0e	61	35	57	b9	86	c1	1d	9e
e	e1	f8	98	11	69	d9	8e	94	9b	1e	87	e9	ce	55	28	df
f	8c	a1	89	0d	bf	e6	42	68	41	99	2d	0f	b0	54	bb	16

Fig. 11 The lookup table (Sbox) in hexadecimal format

The InvSubBytes is obtained by using the inverse table shown below:

x \ y	0	1	2	3	4	5	6	7	8	9	a	b	c	d	e	f
0	52	09	6a	d5	30	36	a5	38	bf	40	a3	9e	81	f3	d7	fb
1	7c	e3	39	82	9b	2f	ff	87	34	8e	43	44	c4	de	e9	cb
2	54	7b	94	32	a6	c2	23	3d	ee	4c	95	0b	42	fa	c3	4e
3	08	2e	a1	66	28	d9	24	b2	76	5b	a2	49	6d	8b	d1	25
4	72	f8	f6	64	86	68	98	16	d4	a4	5c	cc	5d	65	b6	92
5	6c	70	48	50	fd	ed	b9	da	5e	15	46	57	a7	8d	9d	84
6	90	d8	ab	00	8c	bc	d3	0a	f7	e4	58	05	b8	b3	45	06
7	d0	2c	1e	8f	ca	3f	0f	02	c1	af	bd	03	01	13	8a	6b
8	3a	91	11	41	4f	67	dc	ea	97	f2	cf	ce	f0	b4	e6	73
9	96	ac	74	22	e7	ad	35	85	e2	f9	37	e8	1c	75	df	6e
a	47	f1	1a	71	1d	29	c5	89	6f	b7	62	0e	aa	18	be	1b
b	fc	56	3e	4b	c6	d2	79	20	9a	db	c0	fe	78	cd	5a	f4
c	1f	dd	a8	33	88	07	c7	31	b1	12	10	59	27	80	ec	5f
d	60	51	7f	a9	19	b5	4a	0d	2d	e5	7a	9f	93	c9	9c	ef
e	a0	e0	3b	4d	ae	2a	f5	b0	c8	eb	bb	3c	83	53	99	61
f	17	2b	04	7e	ba	77	d6	26	e1	69	14	63	55	21	0c	7d

Fig. 12 The inverted lookup table (InvSbox) in hexadecimal format

In FIPS-197 (AES standard released by NIST) the substitution box and its inverse used in SubBytes are computed and can be used as they are [9].

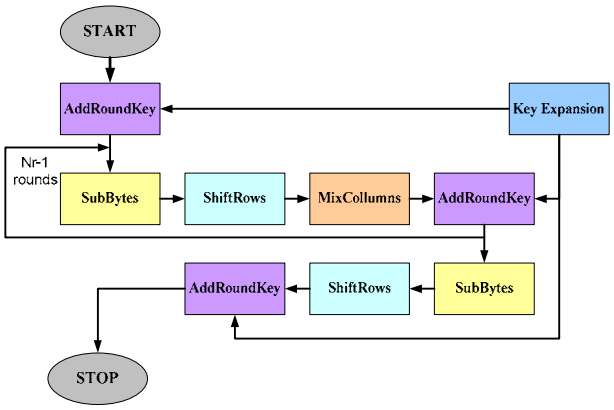

Fig. 13 Rijndael algorithm block scheme

In ShiftRows, the rows of the state are cyclically shifted over different offsets. The first row is not shifted, the second row is shifted over one byte, the third row is shifted over two bytes and, finally, the fourth row is shifted over three bytes.

The MixColumns transformation operates on the State column-by-column, treating each column as a four-term polynomial. The columns are considered as polynomials over GF(2^8) and multiplied modulo x^4+1 with the fixed polynomial: $3x^3+x^2+x+2$.

The AddRoundKey operation consists in the simple addition of the round key with the State. The Round Key is generated by means of the key schedule.
The algorithm uses the cipher key to generate an extended key. This key is used by the AddRoundKey operation. This entire process is called Key Schedule.
Key Schedule consist in two components:

- Key Expansion;
- Round Key selection.

```
KeyExpansion (byte key[4*Nk], word w[Nb*(Nr+1)],
Nk)
        begin
        word temp
        i = 0
        while (i < Nk)
        w[i] = word(key[4*i], key[4*i+1], key[4*i+2],
key[4*i+3])
        i = i+1
end while
        i = Nk
        while (i < Nb * (Nr+1)]
        temp = w[i-1]
        if (i mod Nk = 0)
        temp = SubWord(RotWord(temp)) xor
Rcon[i/Nk]
        else if (Nk > 6 and i mod Nk = 4)
        temp = SubWord(temp)
end if
        w[i] = w[i-Nk] xor temp
        i = i + 1
        end while
end
```

SubWord() is a function that takes a four-byte input word and applies the S-box to each of the four bytes to produce an output word. The function RotWord() takes a word [*a*0,*a*1,*a*2,*a*3] as input, performs a cyclic permutation, and returns the word [*a*1,*a*2,*a*3,*a*0]. The round constant word array, Rcon[i], contains the values given by [*xi-1*,{00},{00},{00}], with *x i*-1 being powers of *x* (*x* is denoted as {02}) in the field GF(2^8), as (*i* starts at 1, not 0).

The first *Nk* words of the expanded key are filled with the Cipher Key. Every following word, w[i], is equal to the XOR of the previous word, w[i-1], and the word *Nk* positions earlier, w[i-*Nk*]. For words in positions that are a multiple of *Nk*, a transformation is applied to w[i-1] prior to the XOR, followed by an XOR with a round constant, Rcon[i]. This transformation consists of a cyclic shift of the bytes in a word (RotWord()), followed by the application of a table lookup to all four bytes of the word (SubWord()).

The Key Expansion routine for 256-bit Cipher Keys (*Nk* = 8) is slightly different than for 128 and 192-bit Cipher Keys. If *Nk* = 8 and i-4 is a multiple of *Nk*, then SubWord() is applied to w[i-1] prior to the XOR.
The algorithm was implemented using three modes of operation: ECB[4], CBC[5], and CFB[6]. In this application the ECB mode was implemented for testing purposes because of the security problems caused by the image redundancy. This mode can only be used in tandem with a compression module.

The encryption and decryption filters contain an interface that the main program uses to connect to these filters and command the encryption keys and the encryption modes. The five keys are built-in and the user can only choose which key to use at a given moment. This solution was chosen to avoid key interception during the transmission. The input and output buffer are twice the size of a frame. The data in processed in RGB mode with a 24 bits colour depth. The secure channel can be created in three ways:

- at the application layer;
- at the network layer;
- at the data link layer.

In this case, the encryption is done at application layer and the only thing that is encrypted is the video transmission.

3.4 RF modules

These modules handle the data link layer. They are built around Winsock 2.0. The receiver filter is responsible with the compensation in the variations of frame rate caused by the auto exposure function of the camera. To do this, a circular buffer was implemented to reassemble the incoming fames. This buffer doesn't forward a frame until it is complete. If only a part of a frame is receiver, the filter waits for the rest before it forwards the frame. The same socket is used for transmitting the changes in encryption modes and encryption keys. The main program access the filter every two seconds through a particular interface and checks for a change. If a change in encryption method or encryption key is detected, the main program commands the change to the decryption filter through another particular interface. In the first frames after the change, the received frames are scrambled. After that the system resynchronises itself. The protocols used for feeding the data to the wireless device were TCP and IP in stream mode. Another way of connecting is the datagram mode, but this mode uses UDP and it's unreliable. The TCP protocol handled packet retransmission and the 802.11 RF module handled error corrections. The radio link could not be used at its full potential because of minimum delay required and the synchronization that was taking place in background. IP is a routable protocol and user at the ground can view transmissions from multiple UAV.

[4] Electronic CodeBook
[5] Cipher Block Chaining
[6] Cipher FeedBack

3.4 Static image encryption example

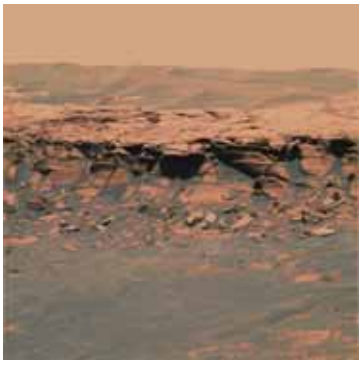

Fig. 14 Original image

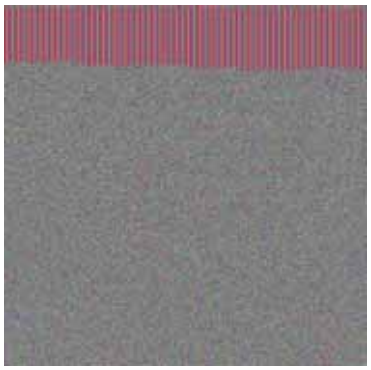

Fig. 15 Encrypted image in ECB mode

Fig. 16 Encrypted image in CBC mode

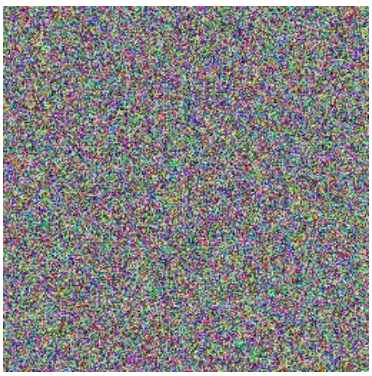

Fig. 17 Encrypted image in CFB mode

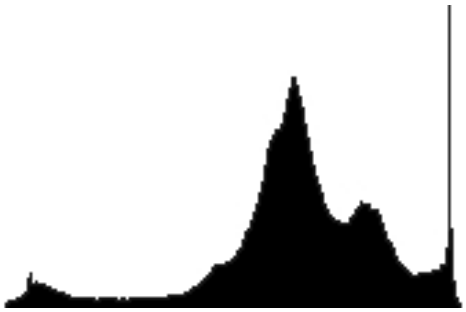

Fig. 18 Histogram of original image

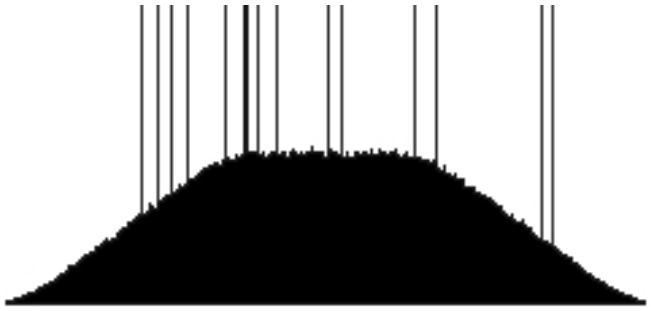

Fig. 19 Histogram of encrypted image in ECB mode

Fig. 20 Histogram of encrypted image in CBC mode

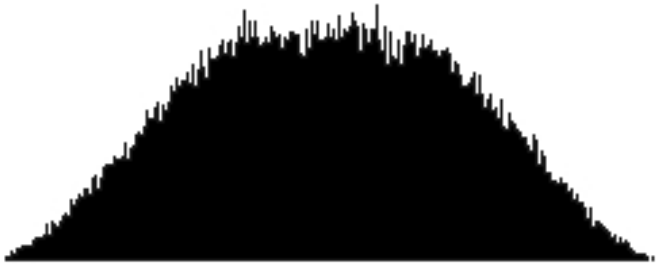

Fig. 21 Histogram of encrypted image in CFB mode

3.5 Real-time encryption example

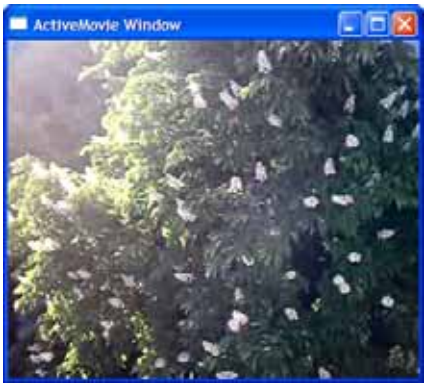

Fig. 22 Plain image

Fig. 23 Encrypted image in ECB mode

Fig. 24 Encrypted image in CBC mode

Fig. 25 Encrypted image in CFB mode

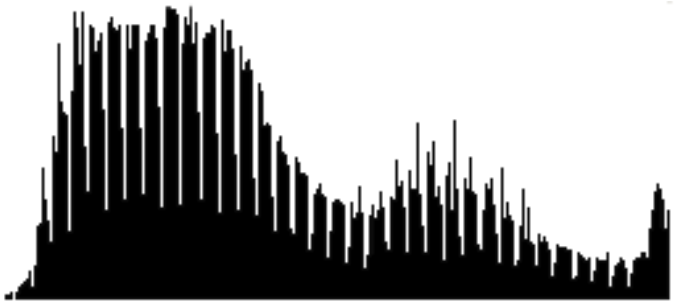

Fig. 26 Histogram of plain image

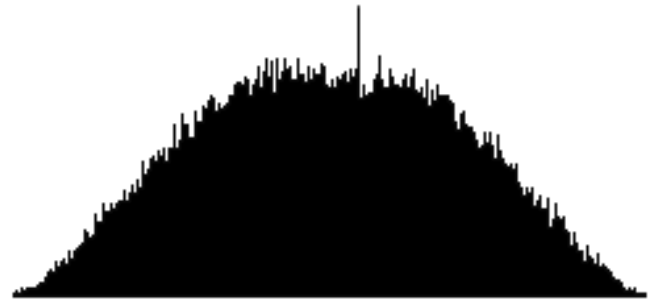

Fig. 27 Histogram of encrypted image in ECB mode

Fig. 28 Histogram of encrypted image in CBC mode

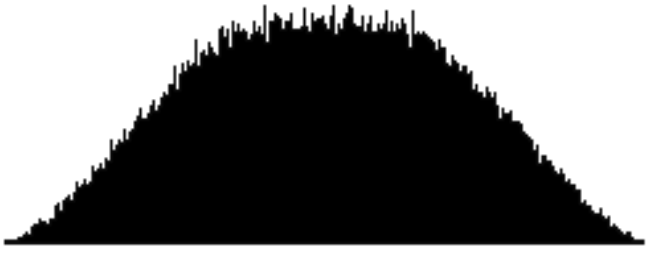

Fig. 29 Histogram of encrypted image in CFB mode

Fig. 30 Plain image

Fig. 31 Encrypted image in ECB mode

Fig. 32 Encrypted image in CBC mode

Fig. 33 Encrypted image in CFB mode

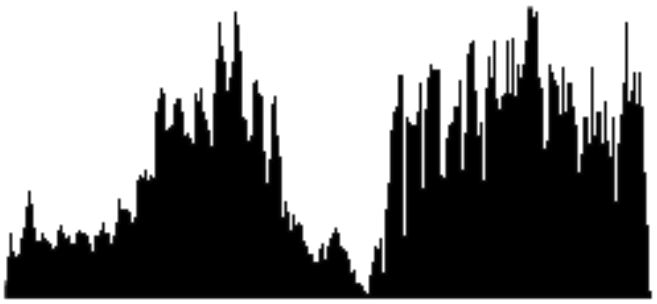

Fig. 34 Histogram of plain image

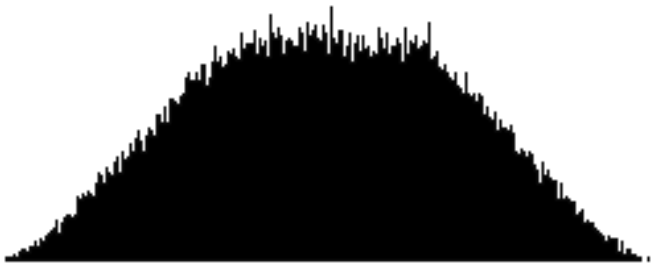

Fig. 35 Histogram of encrypted image in ECB mode

Fig. 36 Histogram of encrypted image in CBC mode

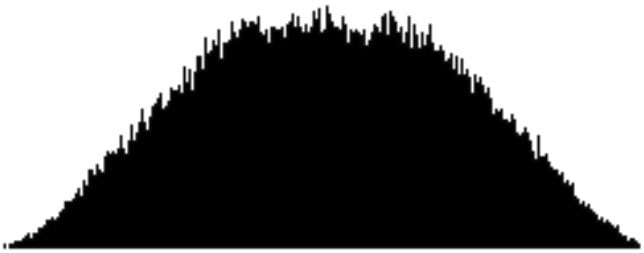

Fig. 37 Histogram of encrypted image in CFB mode

Fig. 38 Plain image

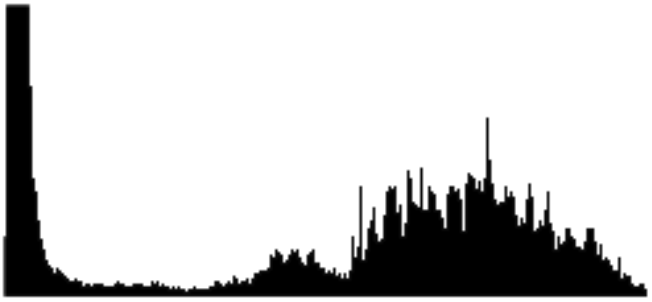

Fig. 42 Histogram of plain image

Fig. 39 Encrypted image in ECB mode

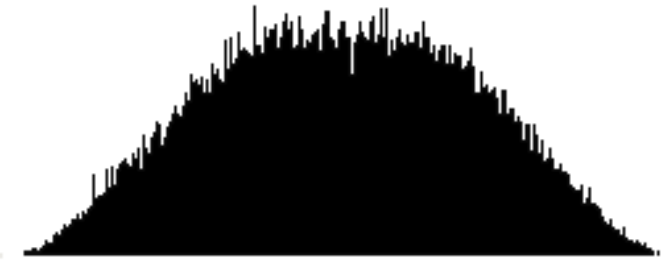

Fig. 43 Histogram of encrypted image in ECB mode

Fig. 40 Encrypted image in CBC mode

Fig. 44 Histogram of encrypted image in CBC mode

Fig. 41 Encrypted image in CFB mode

Fig. 45 Histogram of encrypted image in CFB mode

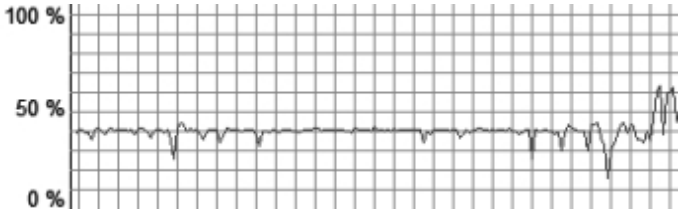

Fig. 46 Link load

Conclusion

In the three modes of operation (ECB, CBC, CFB) the achieved standard deviation is between 49.36 and 49.41. From the statistical point of view de difference is very small. Although ECB mode is the fastest, there are problems in using this mode because it doesn't use the process of chaining and that is why ECB mode has problems; a continuous area of the same color has the same encrypted result [1, 2]. This can be resolved using an image compression module witch eliminates the redundancy [3, 4].

The key length was limited at 128 bits because of the limitations imposed by the processing power and the power consumption. Although the key is small, there are 340282366920938463463374607431768211456 possible combinations. If the key is created using a good pseudo-random generator, the key length shouldn't be a problem [9].

Encryption mode	Average link load	Average frames/second
ECB	39%↔2,304 Mbps	10
CBC	40%↔2,073 Mbps	9
CFB	40%↔2,073 Mbps	9

Table 3 Performance of the encryption modes used

Encryption mode	Std. Dev.
Original image	35.52
ECB	49.39
CBC	49.41
CFB	48.36

Table 4 Standard deviation of the images in different encryption modes

The histograms show that most of the values are gathered near the median.

The encryption algorithm was implemented in software and because of that the frame rate was around 10 frames / second. The VIA C7 microprocessor has a built-in hardware encryption block. The encryption process can be made in hardware at a speed around 2Gbps, leaving enough free resources for image processing tasks, like compression, object recognition, path estimation etc. Although the helicopter runs on batteries, the power consumption of the processor is small (1W in idle mode and 12 W in full load mode).

Real-time encryption example number	Mode	Std. Dev.
1st	Plain image	61.6
	ECB	49.46
	CBC	49.43
	CFB	49.35
2nd	Plain image	67.12
	ECB	49.35
	CBC	49.43
	CFB	49.46
3rd	Plain image	83.85
	ECB	49.31
	CBC	49.36
	CFB	49.24

Table 5 Standard deviations of real-time examples

The histograms show that most of the values are gathered near the median.

The encryption algorithm was implemented in software and because of that the frame rate was around 10 frames / second. The VIA C7 microprocessor has a built-in hardware encryption block. The encryption process can be made in hardware at a speed around 2Gbps, leaving enough free resources for image processing tasks, like compression, object recognition, path estimation etc. Although the helicopter runs on batteries, the power consumption of the processor is small (1W in idle mode and 12 W in full load mode).

There is a background communication between the DirectShow filters. The base system can command the change of encryption modes and encryption keys of the airborne system while running, increasing the strength of the encryption system. The keys are not transmitted via radio link; instead they are built-in and can be programmed before a mission.

DirectShow's big advantage is that the video software chain is modular. In this chain other filters can be added, filters like video compression. The filter chain is constructed and controlled by the main program. If in a moment another type of video processing is needed, the entire chain can be stopped and reconstructed accordingly [7].

The helicopter is silent and small. It can be used in tasks where is necessary to approach the objective without being seen or heard. The disadvantage is that the range is reduced in comparison with a fuel driven helicopter. Being small, it can be transported easily on the field using a small car.

References:

[1] Ciprian Răcuciu, Nicolae Jula, Constantin Bălan, Cosmin Adomnicăi, Aspects of Airborne Continuous Surveillance Systems. Practical Image Encryption Module, *Computational Methods and Inteligent Systems*, WSEAS Press, 2008, pg. 167-172, ISBN 978-960-6766-60-2; ISSN 1790-5117.

[2] Ciprian Răcuciu, Nicolae Jula, Cosmin Adomnicăi, About an image encryption solution adapted for surveillance flying systems, *International Journal Of Mathematical Models And Methods In Applied Sciences*, North Atlantic University Union Press, Issue 1, Volume 2, 2008, pg. 130-135, ISSN 1998-0140.

[3] Ciprian Răcuciu, Nicolae Jula, Florin Marius Pop, Aspects of Mobile Continuous Monitoring Systems. Optimized Image Compression Algorithm. *Circuits, Systems, Signal & Communications,* WSEAS Press, 2008, pg. 181-185, ISBN: 978-960-6766-34-3, ISSN 1790-5117.

[4] Ciprian Răcuciu, Nicolae Jula, Florin-Marius Pop, About an adapted image compression algorithm for surveillance flying systems, *International Journal Of Mathematical Models And Methods In Applied Sciences* Issue 2, Volume 1, 2007, pg. 41- 45, ISSN 1998-0140.

[5] Ciprian Racuciu, *Lecture of Information Theory*, Military Technical Academy Bucharest, 2004.

[6] Romanian Space Agency, AEROSPATIAL program, Contract no. 60/2002, *Electrical Accelerated mini-helicopter used for monitoring.*

[7] Mark D. Pesce, *Programming Microsoft DirectShow for digital video and television*, Microsoft Press, 2003.

[8] Andreas Uhl, Andreeas Pommer, *Image and video encryption - From Digital Rights Management to Secured Personal Communication*, Springer 2004.

[9] Joan Daemen, Vincent Rijmen – *The Design of Rijndael*, Springer, 2002.

[10] Microsoft Platform SDK Help.

[11] A. Menezes, P. van Oorschot, S. Vanstone, *Handbook of Applied Cryptography*, by CRC Press, 1996.

[12] Neal R. Wagner *The Laws of Cryptography with Java Code*, EBook.

[13] Microsoft Windows XP Embedded Help.